

Minion: Unordered Delivery Wire-Compatible with TCP and TLS

Janardhan Iyengar*, Bryan Ford†, Syed Obaid Amin†, Michael F. Nowlan†, Nabin Tiwari*

Draft of 2013/01/13 18:36

Abstract

Internet applications increasingly employ TCP not as a *stream abstraction*, but as a *substrate* for application-level transports, a use that converts TCP’s in-order semantics from a convenience blessing to a performance curse. As Internet evolution makes TCP’s use as a substrate likely to grow, we offer Minion, an architecture for backward-compatible out-of-order delivery atop TCP and TLS. Small OS API extensions allow applications to manage TCP’s send buffer and to receive TCP segments out-of-order. Atop these extensions, Minion builds application-level protocols offering true unordered datagram delivery, within streams preserving strict wire-compatibility with unsecured or TLS-secured TCP connections. Minion’s protocols can run on unmodified TCP stacks, but benefit incrementally when either endpoint is upgraded, for a backward-compatible deployment path. Experiments suggest that Minion can noticeably improve the performance of applications such as conferencing, virtual private networking, and web browsing, while incurring minimal CPU or bandwidth costs.

1 Introduction

TCP [49] was originally designed to offer applications a convenient, high-level communication abstraction with semantics emulating Unix file I/O or pipes. As the Internet has evolved, however, TCP’s original role of offering an *abstraction* has gradually been supplanted with a new role of providing a *substrate* for transport-like, application-level protocols such as SSL/TLS [17], ØMQ [2], SPDY [1], and WebSockets [56]. In this new *substrate* role, TCP’s in-order delivery offers little value since application libraries are equally capable of implementing convenient abstractions. TCP’s strict in-order delivery, however, prevents applications from controlling the *framing* of their communications [14, 19], and incurs a “latency tax” on content whose delivery must wait for the retransmission of a single lost TCP segment.

Due to the difficulty of deploying new transports today [20, 39, 44], applications rarely utilize new out-of-order transports such as SCTP [48] and DCCP [30]. UDP [40] is a popular substrate, but is still not universally supported in the Internet, leading even delay-sensitive applications such as the Skype telephony sys-

tem [6] and Microsoft’s DirectAccess VPN [16], to fall back on TCP despite its drawbacks.

Recognizing that TCP’s use as a substrate is likely to continue and expand, we introduce Minion, a novel architecture for efficient but backward-compatible unordered delivery in TCP. Minion consists of *uTCP*, a small OS extension adding basic unordered delivery primitives to TCP, and two application-level protocols implementing datagram-oriented delivery services that function on either *uTCP* or unmodified TCP stacks.

uTCP addresses delays caused by TCP’s send and receive buffering. On the send side, *uTCP* gives the application a controlled ability to insert data out-of-order into TCP’s send queue, allowing fresh high-priority data to bypass previously-queued low-priority data for example. On the receive side, *uTCP* enables the application to receive out-of-order TCP segments immediately, without delaying their delivery until retransmissions to fill prior holes. Designed for simplicity and deployability, these extensions add less than 600 lines to Linux’s TCP stack.

Minion’s application-level protocols, *uCOBS* and *uTLS*, build general datagram delivery services atop *uTCP* or TCP. A key challenge these protocols address is that TCP offers no reliable out-of-band framing to delimit datagrams in a TCP stream, *uTCP* cannot add out-of-band framing without changing TCP’s wire protocol, and common in-band TCP framing methods assume in-order processing. To make datagrams *self-delimiting* in a TCP stream, *uCOBS* leverages *Consistent Overhead Byte Stuffing* (COBS) [12] to encode application datagrams with at most 0.4% expansion, while reserving a single byte value to delimit encoded datagrams.

Minion adapts the stream-oriented TLS [17] into a secure datagram delivery service atop *uTCP* or TCP. To avoid changing the TLS wire protocol, the *uTLS* receiver heuristically “guesses” TLS record boundaries in stream fragments received out-of-order, then leverages TLS’s cryptographic MAC to confirm these guesses reliably. By preserving strict wire-compatibility with TLS, *uTLS* enables unordered delivery within streams indistinguishable in the network from HTTPS [42], for example.

Experiments with a prototype on Linux show several benefits for applications using TCP. Minion can reduce application-perceived jitter of Voice-over-IP (VoIP) streams atop TCP, and increase perceptible-quality metrics [35]. Virtual private networks (VPNs) that tunnel IP traffic over SSL/TLS, such as OpenVPN [37] or DirectAccess [16], can double the throughput of some tun-

*Franklin and Marshall College, Email: {jiyengar, nabin.tiwari}@fandm.edu

†Yale University, Email: {bryan.ford, michael.nowlan}@yale.edu

neled TCP connections, by employing *u*TCP to prioritize and expedite tunneled ACKs. Web transports can cut the time before a page begins to appear by up to half, achieving the latency benefits of multistreaming transports [19, 33] while preserving the TCP substrate. Use of *u*COBS can incur up to $5\times$ CPU load with respect to raw TCP, due to COBS encoding, but for secure connections, *u*TLS incurs less than 7% CPU overhead (and no bandwidth overhead) atop the baseline cost of TLS.

This paper’s primary contributions are: (a) the first wire-compatible TCP extension we are aware of offering true out-of-order delivery; (b) an API allowing applications to prioritize TCP’s send queue; (c) a novel use of COBS [12] for out-of-order framing atop TCP; (d) an existence proof that out-of-order datagram delivery is achievable from the unmodified, stream-based TLS wire protocol; (e) a prototype and experiments demonstrating Minion’s practicality and performance benefits.

Section 2 motivates Minion by discussing the evolution of TCP’s role in the Internet. Section 3 introduces Minion’s high-level architecture, and Section 4 describes its *u*TCP extensions. Section 5 presents *u*COBS for non-secure datagram delivery, and Section 6 details *u*TLS, a secure analog. Section 7 discusses the current Minion prototype, and Section 8 evaluates its performance experimentally. Section 9 summarizes related work, and Section 10 concludes.

2 Motivating Minion

This section describes how TCP’s role in the network has evolved from a communication *abstraction* to a communication *substrate*, why its in-order delivery model makes TCP a poor substrate, and why other OS-level transports have failed to replace TCP in this role.

2.1 Rise of Application-Level Transports

The transport layer’s traditional role in a network stack is to build high-level communication abstractions convenient to applications, atop the network layer’s basic packet delivery service. TCP’s reliable, stream-oriented design [49] exemplified this principle, by offering an inter-host communication abstraction modeled on Unix pipes, which were the standard *intra-host* communication abstraction at the time of TCP’s design. The Unix tradition of implementing TCP in the OS kernel offered further convenience, allowing much application code to ignore the difference between an open disk file, an intra-host pipe, or an inter-host TCP socket.

Instead of building directly atop traditional OS-level transports such as TCP or UDP, however, today’s applications frequently introduce additional transport-like protocol layers at user-level, typically implemented via application-linked libraries. Examples include the ubiquitous SSL/TLS [17], media transports such as RTP [47],

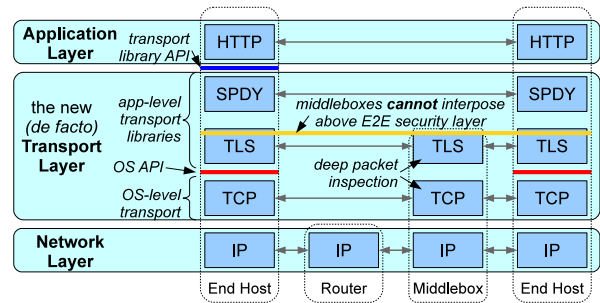


Figure 1: Today’s “*de facto* transport layer” is effectively split between OS and application code.

and experimental multi-streaming transports such as SST [19], SPDY [1], and ØMQ [2]. Applications increasingly use HTTP or HTTPS over TCP as a substrate [39]; this is also illustrated by the W3C’s Web-Socket interface [56], which offers general bidirectional communication between browser-based applications and Web servers atop HTTP and HTTPS.

In this increasingly common design pattern, the “transport layer” as a whole has in effect become a stack of protocols straddling the OS/application boundary. Figure 1 illustrates one example stack, representing Google’s experimental Chromium browser, which inserts SPDY for multi-streaming and TLS for security at application level, atop the OS-level TCP.

One can debate whether a given application-level protocol fits some definition of “transport” functionality. The important point, however, is that today’s applications no longer need, or expect, the underlying OS to provide “convenient” communication abstractions: the application simply links in libraries, frameworks, or middleware offering the abstractions it desires. What today’s applications need from the OS is not convenience, but an *efficient substrate* atop which application-level libraries can build the desired abstractions.

2.2 TCP’s Latency Tax

While TCP has proven to be a popular substrate for application-level transports, using TCP in this role converts its delivery model from a blessing to a curse. Application-level transports are just as capable of sequencing and reassembling packets into a logical data unit or “frame” [14]. By delaying any segment’s delivery to the application until all prior segments are received and delivered, TCP imposes a “latency tax” on all segments arriving within one round-trip time (RTT) after any single lost segment.

Figure 2(a) illustrates TCP’s latency tax, showing cumulative bytes delivered over time during a simple bulk transfer atop TCP versus the unordered *u*TCP substrate we introduce later. The transfer runs on a network path with 60ms RTT, with an unrealistically high 3% loss to

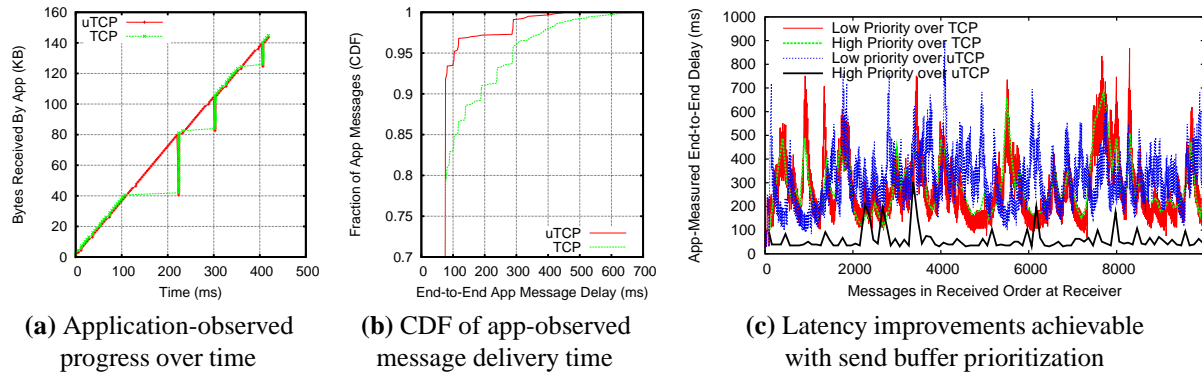


Figure 2: Graphs illustrating some potential benefits from unordered delivery in TCP.

make several losses appear in the short duration. From the application’s perspective, all receive progress stops for one or more round-trip times after any segment’s loss, while *u*TCP continues delivery without interruption.

Figure 2(b) shows a CDF of application-observed delivery latencies in an experiment suggestive of streaming video, where the sender transmits 1448-byte records at fixed 20ms intervals, over a path 100ms RTT and 2% random loss. Atop TCP, over 10% of records are delayed by at least one full RTT, waiting in TCP’s receive queue for a prior segment’s retransmission. Unordered delivery atop *u*TCP significantly delays fewer than 3% of records.

Figure 2(c), showing application-observed message delay over message number, illustrates how *u*TCP’s send-side prioritization affects a series of high-priority application messages interspersed with constant low-priority traffic atop a network-limited TCP connection. In this simple experiment, conducted over a network path with a 60ms RTT and 0.5% loss, unmodified TCP delays high- and low-priority traffic equally, but *u*TCP enables high-priority messages to short-cut the send queue and achieve lower latency.

These experiments merely illustrate TCP’s “latency tax”; we defer to Section 8 for further analysis of TCP’s latency effects under more realistic conditions. For now, the important point is that this latency tax is a fundamental byproduct of TCP’s in-order delivery model, and is irreducible, in that an application-level transport cannot “claw back” the time a potentially useful segment has wasted in TCP’s buffers. The best the application can do is simply to *expect* higher latencies to be common. A conferencing application can use a longer jitter buffer, for example, at the cost of increasing user-perceptible lag. Network hardware advances are unlikely to address this issue, since TCP’s latency tax depends on RTT, which is lower-bounded by the speed of light for long-distance communications.

2.3 Alternative OS-level Transports

All standardized OS-level transports since TCP, including UDP [40], RDP [54], DCCP [30], and SCTP [48], support out-of-order delivery. The Internet’s evolution has created strong barriers against the widespread deployment of new transports other than the original TCP and UDP, however. These barriers are detailed elsewhere [20,39,44], but we summarize two key issues here.

First, since mainstream operating systems offer unprivileged network access only above the transport layer, adding or enhancing a “native” transport built atop IP involves modifying the OS, creating a chicken-and-egg problem for deployment. OS vendors are reluctant to add or modify transports until they perceive widespread demand from applications, but applications are reluctant to rely on new transports without widespread OS support. In effect, it is much more difficult to evolve transport functionality below the red line representing the OS API in Figure 1.

Second, the Internet’s original “dumb network” design, in which routers that “see” only up to the IP layer, has evolved into a “smart network” in which pervasive middleboxes perform deep packet inspection and interposition in transport and higher layers. Firewalls tend to block “anything unfamiliar” for security reasons, often including new transport or application protocols as collateral damage. Network address translators need to rewrite the port numbers in transport layer headers, making them unable to carry traffic for a new or unknown transport without explicit support for that transport. Any packet content not protected by an end-to-end security layer such as TLS—the yellow line in Figure 1—has in effect become “fair game” for middleboxes to inspect and interpose on [41, 55], making it more difficult to evolve transport functionality anywhere below that line.

2.4 Why Not UDP?

As the only widely-supported transport with out-of-order delivery, UDP offers a natural substrate for application-level transports. Even applications otherwise well-suited

to UDP’s delivery model often favor TCP as a substrate, however. A recent study found over 70% of streaming media using TCP [24], and even latency-sensitive conferencing applications such as Skype often use TCP [6].

Network middleboxes support UDP widely but not *universally*. For this reason, latency-sensitive applications seeking maximal connectivity “in the wild” often fall back to TCP when UDP connectivity fails. Skype [6] and Microsoft’s DirectAccess VPN [16], for example, support UDP but can masquerade as HTTP or HTTPS streams atop TCP when needed.

TCP can offer performance advantages over UDP as well. For applications requiring congestion control, an OS-level implementation in TCP may be more timing-accurate than an application-level implementation in a UDP-based protocol, because the OS kernel can avoid the timing artifacts of system calls and process scheduling [58]. Hardware TCP offload engines can optimize common-case efficiency in end hosts [32], and performance enhancing proxies can optimize TCP throughput across diverse networks [11, 13]. Since middleboxes can track TCP’s state machine, they impose much longer idle timeouts on open TCP connections—nominally two hours [23]—whereas UDP-based applications must send keepalives every two minutes to keep an idle connection open [4], draining power on mobile devices.

For applications, TCP versus UDP represents an “all-or-nothing” choice on the spectrum of services applications need. Applications desiring some but not all of TCP’s services, such as congestion control but unordered delivery, must reimplement and tune all other services atop UDP or suffer TCP’s performance penalties.

Without dismissing UDP’s usefulness, we believe the factors above suggest that TCP will remain a popular substrate, even for latency-sensitive applications that can benefit from out-of-order delivery, and that a deployable and backward-compatible workaround to TCP’s latency tax can significantly benefit such applications.

3 Minion Architecture Overview

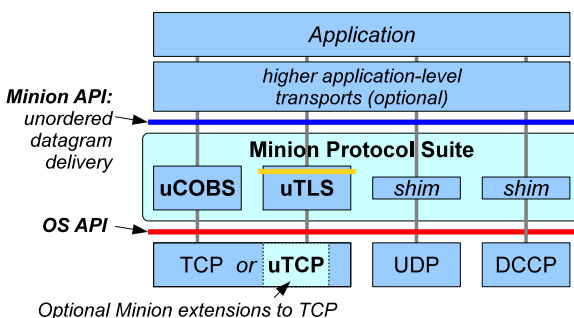


Figure 3: Minion architecture

Minion is an architecture and protocol suite designed to meet the needs of today’s applications for efficient unordered delivery built atop either TCP or UDP. Minion itself offers no high-level abstractions: its goal is to serve applications and higher application-level transports, by acting as a “packhorse” carrying raw datagrams as reliably and efficiently as possible across today’s diverse and change-averse Internet.

3.1 All-Terrain Unordered Delivery

Figure 3 illustrates Minion’s architecture. Applications and higher application-level transports link in and use Minion in the same way as they already use existing application-level transports such as DTLS [43], the datagram-oriented analog of SSL/TLS [17]. In contrast with DTLS’s goal of layering security atop datagram transports such as UDP or DCCP, Minion’s goal is to offer efficient datagram delivery atop *any* available OS-level substrate, including TCP.

While many protocols embed datagrams or application-level frames into TCP streams using delimiting schemes, to our knowledge Minion is the first application-level transport that, under suitable conditions, offers true unordered delivery atop TCP. Minion effectively offers relief from TCP’s latency tax: the loss of one TCP segment in the network no longer prevents datagrams embedded in subsequent TCP segments from being delivered promptly to the application.

3.2 Minion Architecture Components

Minion consists of several application-level transport protocols, together with a set of optional enhancements to end hosts’ OS-level TCP implementations.

Minion’s enhanced OS-level TCP stack, which we call *uTCP* (“unordered TCP”), includes sender- and receiver-side API features supporting unordered delivery and prioritization, detailed in Section 4. These enhancements affect only the OS API through which application-level transports such as Minion interact with the TCP stack, and make *no* changes to TCP’s wire protocol.

Minion’s application-level protocol suite currently consists of the following main components:

- *uCOBS* is a protocol that implements a minimal unordered datagram delivery service atop either unmodified TCP or *uTCP*, using COBS encoding [12] to facilitate out-of-order datagram delimiting and prioritized delivery, as described later in Section 5.
- *uTLS* is a modification of the traditionally stream-oriented TLS [17], offering a secure, unordered datagram delivery service atop TCP or *uTCP*. The wire-encoding of *uTLS* streams is designed to be indistinguishable in the network from conventional, encrypted TLS-over-TCP streams (e.g., HTTPS), offer-

ing a maximally conservative design point that makes no network-visible changes “below the yellow line” in Figure 1. Section 6 describes *uTLS*.

- Minion adds shim layers atop OS-level datagram transports, such as UDP and DCCP, to offer applications a consistent API for unordered delivery across multiple OS-level transports. Since these shims are merely wrappers for OS transports already offering unordered delivery, this paper does not discuss them in detail.

Minion currently leaves to the application the decision of *which* protocol to use for a given connection: e.g., *uCOBS* or *uTLS* atop TCP/*uTCP*, or OS-level UDP or DCCP via Minion’s shims. We are developing an experimental *negotiation protocol* to explore the protocol configuration space dynamically, optimizing protocol selection and configuration for the application’s needs and the network’s constraints [21], but we defer this enhancement to future work. Many applications already incorporate simple negotiation schemes, however—e.g., attempting a UDP connection first and falling back to TCP if that fails—and adapting these mechanisms to engage Minion’s protocols according to application-defined preferences and decision criteria should be straightforward.

3.3 Compatibility and Deployability

Minion addresses the key barriers to transport evolution, outlined in Section 2.3, by creating a backward-compatible, incrementally deployable substrate for new application-layer transports desiring unordered delivery. Minion’s deployability rests on the fact that it can, when necessary, avoid relying on changes either “below the red line” in the end hosts (the OS API in Figure 1), or “below the yellow line” in the network (the end-to-end security layer in Figure 1).

While Minion’s *uCOBS* and *uTLS* protocols offer maximum performance benefits from out-of-order delivery when both endpoints include OS support for Minion’s *uTCP* enhancements, *uCOBS* and *uTLS* still function and interoperate correctly even if neither endpoint supports *uTCP*, and the application need not know or care whether the underlying OS supports *uTCP*. If only one endpoint OS supports *uTCP*, Minion still offers incremental performance benefits, since *uTCP*’s sender-side and receiver-side enhancements are independent. A *uCOBS* or *uTLS* connection atop a mixed TCP/*uTCP* endpoint pair benefits from *uTCP*’s sender-side enhancements for datagrams sent by the *uTCP* endpoint, and the connection benefits from *uTCP*’s receiver-side enhancements for datagrams received by the *uTCP* endpoint.

Addressing the challenge of network-compatibility with middleboxes that filter new OS-level transports and sometimes UDP, Minion offers application-level trans-

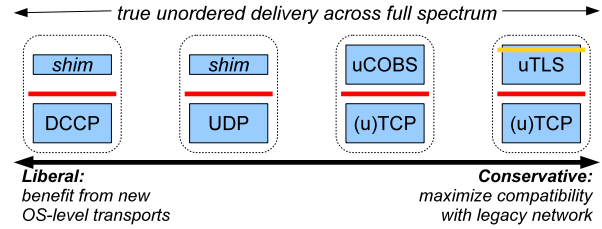


Figure 4: Continuum of Minion configuration tradeoffs

ports a continuum of substrates representing different tradeoffs between suitability to the application’s needs and compatibility with the network. Figure 4 illustrates this continuum of Minion configurations. At the “liberal” end, an application can still use whatever OS-level transport is most suited to its purposes, such as UDP, DCCP [30], or SCTP [48], for paths on which these unordered OS-level transports operate.

On the “moderate conservative” side, *uCOBS* offers an unordered substrate wire-compatible with TCP up through the OS-level transport layer. At the “extreme conservative” end, *uTLS* offers an unordered substrate wire-compatible not only with TCP, but with the ubiquitous TLS-over-TCP streams on which HTTPS (and hence Web security and E-commerce) are based, likely to operate in almost any network environment purporting to offer “Internet access.”

A final issue is compatibility with existing applications. Since most of Minion operates at application-level, applications must be changed to use the Minion API. A pair of application endpoints may also need to negotiate whether to use Minion, or to run directly atop OS-level transports for compatibility with earlier versions of the application. This challenge is comparable to the cost of adding TLS or DTLS support to an application, and the popularity of application-level transports such as TLS suggests that these costs are surmountable. Minion’s application-level functionality might eventually be merged into existing or future application-level transports and communication frameworks, making its benefits available with few or no application changes.

4 *uTCP*: Unordered TCP

Minion enhances the OS’s TCP stack with API enhancements supporting unordered delivery in both TCP’s send and receive paths, enabling applications to reduce transmission latency at both the sender- and receiver-side end hosts when both endpoints support *uTCP*. Since *uTCP* makes no changes to TCP’s wire protocol, two endpoints need not “agree” on whether to use *uTCP*: one endpoint gains latency benefits from *uTCP* even if the other endpoint does not support it. Further, an OS may choose independently whether to support the sender- and receiver-

side enhancements, and when available, applications can activate them independently.

In this spirit of Section 2, *uTCP* does *not* seek to offer “convenient” or “clean” unordered delivery abstractions directly at the OS API. Instead, *uTCP*’s design is motivated by the goals of maintaining exact compatibility with TCP’s existing wire-visible protocol and behavior, and facilitating deployability by minimizing the extent and complexity of changes to the OS’s TCP stack. The design presented here is only one of many viable approaches, with different tradeoffs, to supporting unordered delivery in TCP. Section 4.3 briefly outlines a few such alternatives.

uTCP focuses on reducing latency only at the end hosts. There are many ways to reduce TCP latency in the network, such as using delay-based or explicit congestion control [7, 28]. These valuable efforts are independent of and complementary to Minion’s goals and are not addressed here.

We describe *uTCP*’s API enhancements in terms of the BSD sockets API, although *uTCP*’s design contains nothing inherently specific to this API.

4.1 Receiver-Side Enhancements

uTCP adds one new socket option affecting TCP’s receive path, enabling applications to request immediate delivery of TCP segments received out of order. An application opens a TCP stream the usual way, via `connect()` or `accept()`, and may use this stream for conventional in-order communication before enabling *uTCP*. Once the application is ready to receive out-of-order data, it enables the new option `SO_UNORDERED` via `setsockopt()`, which changes TCP’s receive-side behavior in two ways.

First, whereas a conventional TCP stack delivers received data to the application only when prior gaps in the TCP sequence space are filled, the *uTCP* receiver makes data segments available to the application immediately upon receipt, skipping TCP’s usual reordering queue. The application obtains this data via `read()` as usual, but the first data byte returned by a `read()` call may no longer be the one logically following the last byte returned by the prior `read()` call, in the byte stream transmitted by the sender. The data the *uTCP* stack delivers to the application in successive `read()` calls may skip forward and backward in the transmitted byte stream, and *uTCP* may even deliver portions of the transmitted stream multiple times. *uTCP* guarantees only that the data returned by one `read()` call corresponds to *some* contiguous sequence of bytes in the sender’s transmitted stream, and that barring connection failure, *uTCP* will *eventually* deliver every byte of the transmitted stream at least once.

Second, when servicing an application’s `read()` call, the *uTCP* receiver prepends a short header to the returned data, indicating the logical offset of the first returned byte in the sender’s original byte stream. The *uTCP* stack computes this logical offset simply by subtracting the Initial Sequence Number (ISN) of the received stream from the TCP sequence number of the segment being delivered. Using this metadata, the application can piece together data segments from successive `read()` calls into longer contiguous *fragments* of the transmitted byte stream.

Figure 5 illustrates *uTCP*’s receive-side behavior, in a simple scenario where three TCP segments arrive in succession: first an in-order segment, then an out-of-order segment, and finally a segment filling the gap between the first two. With *uTCP*, the application receives each segment as soon as it arrives, along with the sequence number information it needs to reconstruct a complete internal view of whichever fragments of the TCP stream have arrived.

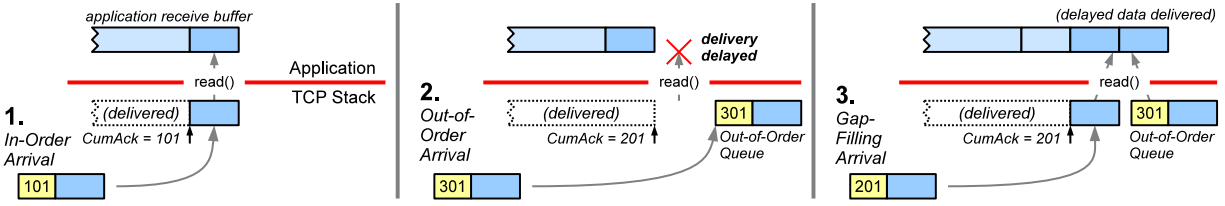
The *uTCP* receiver retains in its receive buffer the TCP headers of segments received and delivered out-of-order, until its cumulative acknowledgement point moves past these segments, and generates acknowledgments and selective acknowledgments (SACKs) exactly as TCP normally would. The *uTCP* receiver does not increase its advertised receive window when it delivers data to the application out-of-order, so the advertised window tracks the cumulative in-order delivery point exactly as in TCP. In this fashion, *uTCP* maintains wire-visible behavior identical to TCP while delivering segments to the application out-of-order.

The *uTCP* receive path assumes the sender may be an unmodified TCP, and TCP’s stream-oriented semantics allow the sending TCP to segment the sending application’s stream at arbitrary points—independent of the boundaries of the sending application’s `write()` calls, for example. Further, network middleboxes may silently *re-segment* TCP streams, making segment boundaries observed at the receiver differ from the sender’s original transmissions [26]. An application using *uTCP*, therefore, must not assume anything about received segment boundaries. This is a key technical challenge to using *uTCP* reliably, and addressing this challenge is one function of *uCOBS* and *uTLS*, described later.

4.2 Sender-Side Enhancements

While *uTCP*’s receiver-side enhancements address the “latency tax” on segments waiting in TCP’s reordering buffer, TCP’s sender-side queue can also introduce latency, as segments the application has already written to a TCP socket—and hence “committed” to the network—wait until TCP’s flow and congestion control allow their transmission. Many applications can benefit from the

(a) Delivery in standard TCP



(b) Delivery in uTCP

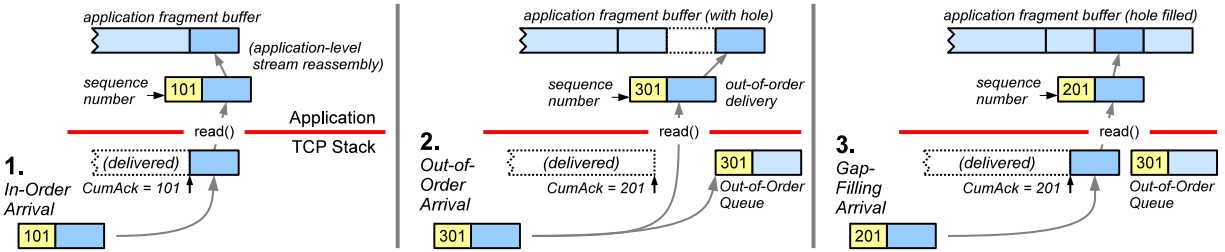


Figure 5: Delivery behavior of (a) standard TCP, and (b) uTCP, upon receipt of in-order and out-of-order segments.

ability to “late-bind” their decision on *what* to send until the last possible moment. An application-level multi-streaming transport with prioritization, such as SST [19] or SPDY [1], would prefer high-priority packets not to get “stuck” behind low-priority packets in TCP’s send queue. In applications such as games and remote-access protocols, where the receiver typically desires *only the freshest* in a stream of real-time status updates, the sender would prefer that new updates “squash” any prior updates still in TCP’s send queue and not yet transmitted.

The Congestion Manager architecture [5] addressed this desire to “late-bind” the application’s transmission decisions, by introducing an upcall-based API in which the OS performs no send buffering, but instead signals the application whenever the application is permitted to send. Upcalls represent a major change to conventional sockets APIs, however, and introduce issues such as how the OS should handle an application that fails to service an upcall promptly, leaving its allocated transmission time-slot unfilled yet unavailable to competing applications waiting to send.

In the spirit of maximizing deployability, uTCP adopts a more limited but less invasive design, by retaining TCP’s send buffer but giving applications some control over it. After enabling uTCP’s new socket option `SO_UNORDEREDSEND`, the OS expects any subsequent `write()` to that socket to include a short header, containing metadata that uTCP reads and strips before placing the remaining data on TCP’s send buffer. The uTCP header contains an integer *tag* and a set of optional flags controlling uTCP’s send-side behavior.

By default, uTCP interprets tags as priority levels. Instead of unconditionally placing the newly-written data at the tail of the send queue as TCP normally would,

uTCP *inserts* the newly-written data into the send queue *just before* any lower-priority data in the send queue and not yet transmitted. The application thus avoids higher-priority packets being delayed by lower-priority packets enqueued earlier, while the OS avoids the complexity and security challenges of an upcall API.

For strict TCP wire-compatibility, uTCP never inserts new data into the send queue ahead of any previously-written data that has ever been transmitted in whole or in part: e.g., ahead of data from a prior application write already partly transmitted and awaiting acknowledgement. If an application writes a large low-priority buffer, then writes higher-priority data after transmission of the low-priority data has begun, uTCP inserts the high-priority data after the entire low-priority write and never in the middle. This constraint enables the application to control the boundaries on which send buffer reordering is permitted, independent of the current MTU and TCP segmentation behavior.

A simple uTCP refinement, which we intend to explore in future work, is to include a *squash* flag in the metadata header the application prepends to each write. If set, while inserting the newly-written data in tag-priority order, uTCP would also remove and discard any data previously written with exactly the same tag, that has not yet been transmitted in whole or in part. This refinement would enable update-oriented applications such as games to avoid the bandwidth cost transmitting old updates superseded by newer data.

4.3 Design Alternatives

uTCP pursues a conservative point in a large design space, in which many alternatives present interesting tradeoffs. We briefly examine a few such alternatives.

Disable TCP congestion control By default, *u*TCP makes no changes to TCP congestion control, making *u*TCP’s unordered delivery model more directly comparable to DCCP [30] than to UDP [40]. Some applications may prefer a UDP-like substrate without congestion control: e.g., media applications that adapt to congestion by transitioning between a set of fixed bit-rates. Purely for experimentation, we extended TCP’s API to enable the application to disable congestion control at the sender per-connection—as Linux has already supports per-interface [25]—offering applications more UDP-like behavior. Such extensions are independent and orthogonal to the unordered delivery extensions we focus on here, however, and we take no position on the policy question of whether the OS *should* allow applications to disable congestion control.

Assign Sequence Numbers at Write Time All application-directed reordering of the send buffer currently occurs *before* *u*TCP assigns sequence numbers to segments. Thus, TCP sequence numbers appearing on the wire still reflect transmission order from the sending *host*, exactly preserving TCP’s wire-visible behavior. An alternative would be for *u*TCP to assign TCP sequence numbers immediately as the application *writes* to the send buffer, preserving these sequence numbers through send buffer reordering. This alternative could simplify some applications, as discussed later in Section 6.1. Send buffer reordering would instead mimic reordering in the network, however, which could violate the common assumption that reordering in the network is rare. Frequent reordering due to send-side prioritization could cause TCP segments to hit “slow paths” in middleboxes and the receiving TCP stack more frequently, for example, and could interfere with the sender’s congestion control by creating the illusion of false loss events, unless the congestion control scheme is adjusted to compensate. Assigning sequence numbers immediately upon write would also preclude the *squash* flag refinement above.

Disable retransmission of old TCP segments The *u*TCP receiver could always proactively move its cumulative acknowledgment point past *all* received segments, including those received out-of-order, effectively pretending that lost segments were received and tricking the sender into not retransmitting them. This change would enable applications to avoid the bandwidth cost of retransmitting obsolete data. As a side-effect, this change would interfere with the sender’s congestion control by effectively hiding all “loss events” other than retransmission timeouts, which might or might not be desirable. This change would also affect wire-visible behavior, however: middleboxes could observe the receiver acknowledging TCP sequence number “holes” whose data

was dropped upstream from the middlebox, resulting in unpredictable middlebox behavior [26].

Send new data in retransmission segments *u*TCP could send new application data, instead of the data originally sent, in retransmissions of unacknowledged sequence ranges. This change would eliminate the cost of retransmitting obsolete data, and the need for the sending TCP to buffer transmitted data until acknowledgment. Network delays, reordering, and segmentation could result in the receiver seeing a mix of “new” and “old” data spliced at arbitrary boundaries, however, and inconsistent retransmission can trigger unpredictable middlebox behavior [26].

Increase the receive window on out-of-order delivery The *u*TCP receiver technically need not buffer out-of-order data once it has been delivered to the application—only the sequence ranges it needs to send correct ACKs and SACKs. Since the TCP receive window announcement traditionally reflects the amount of buffer space available at the receiver, the *u*TCP receiver could increase its receive window announcement when it delivers out-of-order data, as TCP does when it delivers in-order data. This change may introduce a denial-of-service attack vulnerability, however, where a sender keeps sending data out-of-order to a *u*TCP stack indefinitely without ever going back to “fill the gaps” and advance the cumulative acknowledgment point, leading to unbounded state in the *u*TCP stack and possibly in the application. In *u*TCP’s more conservative design, the receive window imposes a limit on the number of out-of-order bytes outstanding before the sender must retransmit lost data and move the cumulative-acknowledgment point forward.

Offer the application exactly-once data delivery *u*TCP could guarantee the delivery of a given byte of the transmitted stream *exactly* once, rather than at least once, by “pruning” data already received from subsequent out-of-order and in-order deliveries. Unlike the above design changes, this one would not affect network-visible behavior and may be worthwhile. But it would increase the complexity of *u*TCP’s modifications to the TCP stack, for dubious benefit to the application. A *u*TCP application must in any case contain the state and logic necessary to assemble segments received out-of-order into larger fragments and scan these fragments for useful application records. On top of this, we find the incremental complexity cost of detecting and ignoring duplicate data at application level to be small.

5 *u*COBS: Simple Datagrams on TCP

Since *u*TCP’s design attempts to minimize OS changes, its unordered delivery primitives do not directly offer applications a convenient, general-purpose datagram substrate. Minion’s *u*COBS protocol bridges this semantic

gap, building atop *u*TCP (or standard TCP) a lightweight datagram delivery service comparable to UDP or DCCP. This first section first introduces the challenge of delimiting datagrams, then presents *u*COBS' solution and discusses alternatives.

5.1 Self-Delimiting Datagrams for *u*TCP

Applications built on datagram substrates such as UDP generally assume the underlying layer preserves datagram boundaries. If the network fragments a large UDP datagram, the receiving host reassembles it before delivery to the application, and a correct UDP never merges multiple datagrams, or datagram fragments, into one delivery to the receiving application. TCP's stream-oriented semantics do not preserve any application-relevant frame boundaries within a stream, however. Both the TCP sender and network middleboxes can and do coalesce TCP segments or re-segment TCP streams in unpredictable ways [26]. Conventional TCP applications, which send and receive TCP data in-order, commonly address this issue by delimiting application-level frames with some length-value encoding, enabling the receiver to locate the next frame in the stream from the previous frame's position and header content.

Since *u*TCP's receive path effectively just bypasses TCP's reordering buffer, delivering received segments to the application as they arrive, a stream fragment received out-of-order from *u*TCP may begin at any byte offset in the stream, and not at a frame boundary meaningful to the application. Since the receiver is by definition missing some data sent prior to this out-of-order segment, it cannot rely on preceding stream content to compute the next frame's position.

Reliable use of *u*TCP, therefore, requires that frames embedded in the TCP stream be *self-delimiting*: recognizable without knowledge of preceding or following data. There are two common ways to make frames in a byte stream self-delimiting: the sender and receiver can agree on frame boundaries in advance, or arrange for some distinguished byte or sequence to appear in the stream *only* at frame boundaries.

Fixed-Length Frames: The first approach is practical atop *u*TCP if all application frames are fixed size, since *u*TCP's receive path prepends a header to each received segment containing segment's logical byte offset in the TCP stream. If *u*TCP delivers a fragment with stream offset s , and all frames are f bytes in size, the next frame boundary starts $f - ((s - 1) \bmod f) - 1$ bytes into the received fragment.

Most applications expect a datagram substrate to support variable-size frames, however. Even in media applications, whose codecs may support a constant-bit-rate (CBR) mode, variable- and average-bit-rate modes are commonly desired. Even teleconferencing applications

that use CBR mode often detect and encode "silence" in special, small frames. To avoid wasting bandwidth by padding all frames to a fixed size, we wish *u*COBS to support variable-size frames.

Variable-Length Frames: If the application-level frames happen to be encoded so as never to include some "reserved" byte value, such as zero, then we could use that byte reserved value to delimit frames within *u*TCP streams. Since we wish *u*COBS to support general-purpose delivery of datagrams of variable length containing arbitrary byte values, however, *u*COBS must explicitly (re-)encode the application's datagrams in order to reserve some byte value to serve as a delimiter.

Any scheme that encodes arbitrary byte streams into strings utilizing fewer than 256 symbols will serve this purpose, such as the ubiquitous *base64* scheme, which encodes byte streams into strings utilizing only 64 ASCII symbols plus whitespace. Since *base64* encodes three bytes into four ASCII symbols, however, it expands encoded streams by a factor of $4/3$, incurring a 33% bandwidth overhead. Since *u*COBS needs to reserve only *one* byte value for delimiters, and not the large set of byte values considered "unsafe" in E-mail or other text-based message formats, *base64* encoding is unnecessarily conservative for *u*COBS' purposes.

5.2 Operation of *u*COBS

To encode application datagrams efficiently, *u*COBS employs *consistent-overhead byte stuffing*, or COBS [12]. COBS is analogous to *base64*, except that it encodes byte streams to reserve only *one* distinguished byte value (e.g., zero), and utilizes the remaining 255 byte values in the encoding. COBS could in effect be termed "base255" encoding. By reserving only one byte value, COBS incurs an expansion ratio of at most $255/254$, or 0.4% bandwidth overhead.

Transmission: When an application sends a datagram, *u*COBS first COBS-encodes the datagram to remove all zero bytes. *u*COBS then prepends a zero byte to the encoded datagram, appends a second zero byte to the end, and writes the encoded and delimited datagram to the TCP socket. Since this sender-side encoding and transmission process operates entirely at application level within *u*COBS, and does not rely on any OS-level extensions on the sending host, *u*COBS operates even if the sender-side OS does not support *u*TCP.

The application can assign priorities to datagrams it submits to *u*COBS, however, and if the sender's OS does support the *u*TCP extensions in Section 4.2, *u*COBS pass these priorities to the *u*TCP sender, enabling higher-priority datagrams to pass lower-priority datagrams already enqueued. Since *u*TCP respects application `write()` boundaries while reordering the send

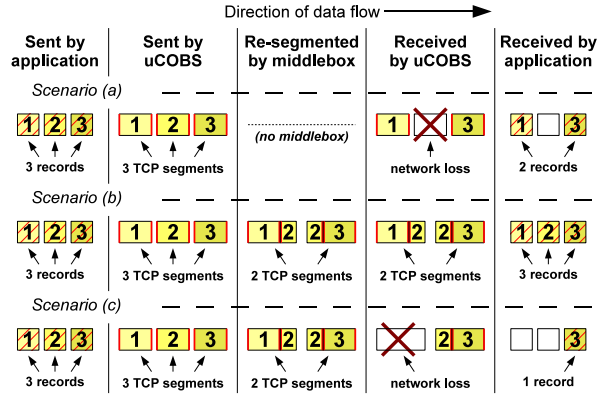


Figure 6: An example illustrating a *uCOBS* transfer

queue, *uCOBS* preserves its delimiting invariant simply by writing each encoded datagram—with the leading and trailing zero bytes—in a single write.

Reception: At stream creation time, *uCOBS* enables *uTCP*’s receive-side extensions if available. If the receive-side OS does not support *uTCP*, then *uCOBS* simply falls back on the standard TCP API, receiving, COBS-decoding, and delivering datagrams to the application in the order they appear in the TCP sequence space. (This may not be the application’s original send order if the send-side OS supports *uTCP*.)

If the receive-side OS supports *uTCP*, then *uCOBS* receives segments from *uTCP* in whatever order they arrive, then fits them together using the metadata in *uTCP*’s headers to form contiguous fragments of the TCP stream. The arrival of a TCP segment can cause *uCOBS* to create a new fragment, expand an existing fragment at the beginning or end, or “fill a hole” between two fragments and merge them into one. The portion of the TCP stream before the receiver’s cumulative-acknowledgment point, containing no sequence holes, *uCOBS* treats as one large “fragment.” *uCOBS* scans the content of any new, expanded, or merged fragment for properly delimited records not yet delivered to the application. *uCOBS* identifies a record by the presence of two marker bytes surrounding a contiguous sequence of bytes containing no markers or holes. Once *uCOBS* identifies a new record, it strips the delimiting markers, decodes the COBS-encoded content to obtain the original record data, and delivers the record to the application.

5.3 Why Two Markers Per Datagram?

For correctness alone, *uCOBS* need only prepend *or* append a marker byte to each record—not both—but such a design could reduce performance by eliminating opportunities for out-of-order delivery. Consider Scenario (a) in Figure 6, in which an application sends three records. *uCOBS* encodes these records and sends them via three `write()` calls, which TCP in turn sends in three sep-

arate TCP segments. In this scenario, no middleboxes re-segment the TCP stream in the network, but the middle segment is lost. If the *uCOBS* sender were only to *prepend* a marker at the start of each record, the *uCOBS* receiver could not deliver record 1 immediately on receipt, since it cannot tell if record 1 extends into the following “hole” in sequence space. Similarly, if the sender were only to *append* a marker at the end of each record, then *uCOBS* could not deliver segment 3 immediately on receipt, since record 3 might extend backwards into the preceding hole. By adding markers to both ends of each record, *uCOBS* ensures that the receiver can deliver each record as soon as all of its segments arrive.

These markers enable *uCOBS* to offer reliable out-of-order delivery even if network middleboxes re-segment the TCP stream. In Scenario (b) in Figure 6, for example, *uCOBS* sends three records encoded into three TCP segments as above, but a middlebox re-segments them into two longer TCP segments, whose boundary splits record 2 into two parts. If neither of these segments are lost, then the *uCOBS* receiver can deliver record 1 immediately upon receipt of the first TCP segment, and can deliver records 2 and 3 upon receipt of the second segment. If the first segment is lost as shown in Scenario (c), however, the *uCOBS* receiver cannot deliver the missing record 1 or the partial record 2, but can still deliver record 3 as soon as the second TCP segment arrives.

6 *uTLS*: Secure Datagrams on TCP

While *uCOBS* offers out-of-order record delivery wire-compatible up to the TCP level, middleboxes often perform deep packet inspection and even manipulation on the *content* of TCP streams as well [41, 55]. An increasingly *de facto* rule is that anything *not* encrypted in a TCP or UDP stream is “fair game” for middleboxes. An application’s only way to ensure “end-to-end” communication in practice, therefore, is via end-to-end encryption and authentication. But network-layer mechanisms such as IPsec [29] face the same deployment challenges as new secure transports [19], and remain confined to the niche of corporate VPNs. Even VPNs are shifting from IPsec toward HTTPS tunnels [16], the only form of end-to-end encrypted connection almost universally supported on today’s Internet. A network administrator or ISP might disable nearly any other port while claiming to offer “Internet access,” but would be hard-pressed to disable SSL/TLS on port 443, the foundation for E-commerce.

We could layer TLS atop *uCOBS*, but TLS decrypts and delivers data only in-order, negating *uTCP*’s benefit. We could also layer DTLS [43], the datagram-oriented version of TLS, atop *uCOBS*, but the resulting wire format—DTLS-encrypted *then* COBS-encoded—would be radically different from TLS over TCP, likely drawing

suspicion from middleboxes especially if used on port 443. The goal of *u*TLS, therefore, is to coax out-of-order delivery from the *existing* TCP-oriented TLS wire format, producing an encrypted datagram substrate indistinguishable on the wire from standard TLS connections (except via analysis of “side-channels” such as packet length and timing, which we do not address).

6.1 Design of *u*TLS

TLS [17] already breaks its communication into *records*, encrypts and authenticates each record, and prepends a header for transmission on the underlying TCP stream. TLS was designed to decrypt records strictly in-order, however, creating three challenges for *u*TLS:

- **Locating record headers out-of-order.** Since encrypted data may contain any byte sequence, there is no reliable way to differentiate a TLS header from record data in the TCP stream, as COBS encoding provides.
- **Encryption state chaining.** Some TLS ciphersuites chain encryption state across records, making records indecipherable until prior records are processed.
- **Record numbers used in MAC computation.** TLS includes a record number, which increases by 1 for each record, in computing the record’s MAC. But the *u*TLS receiver may not know an out-of-order record’s number: holes in TCP sequence space before the record could contain an unknown number of prior records.

To locate records out-of-order, *u*TLS first scans a received stream fragment for byte sequences that *may* represent a TLS header: i.e., containing a plausible record type, version, and length. While this scan may yield false positives, *u*TLS verifies the inferred header by attempting to decrypt and authenticate the record. If the cryptographic MAC check fails, instead of aborting the connection as TLS normally would, *u*TLS assumes a false positive and continues scanning.

Since TLS’s MAC is designed to prevent resourceful adversaries from constructing a byte sequence the receiver could misinterpret as a record, and it is by definition at least as hard to find such a sequence “accidentally” as to forge one maliciously, TLS security should protect equally well against accidental false positives. One exception is when TLS is using its “null ciphersuite,” which performs no packet authentication. With this ciphersuite, normally used only during initial key negotiation, *u*TLS disables out-of-order delivery to avoid the risk of accepting and delivering false records.

The only obvious solution to the second challenge above is to avoid ciphersuites that chain encryption state across records. Most ciphersuites before TLS 1.1 chain encryption state, unfortunately. Any stream cipher inherently does so, such as the RC4 cipher used in early SSL versions. Most recent ciphersuites use block ciphers

in CBC mode, which does not inherently depend on chained encryption state, but does require an Initialization Vector (IV) for each record. Until recently, TLS produced each record’s IV implicitly from the prior record’s encryption state, making records interdependent.

To fix a security issue, however, TLS 1.1 block ciphers use explicit IVs, which the sender generates independently for each record and prepends to the record’s ciphertext. As a side-effect, TLS 1.1 block ciphers support out-of-order decryption. Since TLS supports negotiation of versions and ciphersuites, *u*TLS simply leverages this process. An application can insist on TLS 1.1 with a block cipher to ensure out-of-order delivery support, or it can permit older ciphersuites to maximize interoperability, at the risk of sacrificing out-of-order delivery.

The third challenge is the implicit “pseudo-header” TLS uses in computing the MAC for each packet. This pseudo-header includes a “sequence number” that TLS increments once per *record*, rather than per *byte* as with TCP sequence numbers. When *u*TLS identifies a possible TLS record in a TCP fragment received out-of-order, the receiver knows only the byte-oriented TCP stream offset, and not the TLS record number. Since records are variable-length, unreceived holes prior to a record to be authenticated may “hide” a few large records or many smaller records, leaving the receiver uncertain of the correct record number for the MAC check.

To authenticate records out-of-order without modifying the TLS ciphersuite, therefore, *u*TLS attempts to *predict* the record’s likely TLS record number, using heuristics such as the average size of past records, and may try several adjacent record numbers to find one for which the MAC check succeeds. If *u*TLS fails to find a correct TLS record number, it cannot deliver the record out-of-order, but will still eventually deliver the record in-order.

The current *u*TLS supports only receiver-side unordered delivery, and not the send-side *u*TCP enhancements in Section 4.2, because send-side reordering complicates record number prediction. A future enhancement we intend to explore is for *u*TLS to prepend an explicit record number to application payloads before encryption. Since encryption does not depend on record number, the receiving *u*TLS can decrypt the record number for use in the MAC check, avoiding the need to predict record numbers and enabling send-side reordering. Since the only wire-protocol change is protected by encryption, the change would be invisible to middleboxes. Preserving end-to-end backward compatibility may require a way to negotiate “under encryption” the use of explicit record numbers, however.

7 Prototype Implementation

This section describes the current Minion prototype, which implements *u*TCP in the Linux kernel, and imple-

ments *uCOBS* and *uTLS* in application-linked libraries. The *uTCP* prototype is Linux-specific, but we expect the API extensions it implements and the application-level libraries to be portable.

The *uTCP* Receiver in Linux: The *uTCP* prototype adds about 240 lines and modifies about 50 lines of code in the Linux 2.6.34 kernel, to support the new `SO_UNORDERED` socket option. This extension involved two main changes. First, *uTCP* modifies the TCP code that delivers segments to the application, to prepend a 5-byte metadata header to the data returned from each `read()` system call. This header consists of a 1-byte flags field and a 4-byte TCP sequence number. One flag bit is currently used, with which *uTCP* indicates whether it is delivering data in-order or out-of-order. Second, if TCP’s in-order queue is empty, *uTCP*’s `read()` path checks and returns data from the out-of-order queue. To minimize kernel changes, segments remain in the out-of-order queue after delivery, so *uTCP* will eventually deliver the same data again in-order.

The *uTCP* Sender in Linux: On the send path, *uTCP* adds about 250 lines of kernel code and modifies about 20 in Linux 2.6.34, supporting a new `SO_UNORDEREDSEND` socket option via two changes.

First, *uTCP* expects the application to prepend a 5-byte header, containing a 1-byte flags field and a 4-byte tag, to the data passed to each `write()`. The flags are currently unused, and the tag indicates message priority.

Second, *uTCP* inserts the data from each `write()` into the kernel’s send queue in priority order. Linux’s TCP send queue is a simple FIFO that packs application data into kernel buffers sized to the TCP connection’s Maximum Segment Size (MSS). When inserting application messages non-sequentially, however, *uTCP* must preserve application message boundaries in the kernel. For simplicity, *uTCP* allocates kernel buffers (`skbuffs`) so that each message sent via *uTCP* starts a new `skbuff`, and may span several `skbuffs`, but no `skbuff` contains data from multiple application writes.

Disabling Linux’s usual packing of MSS-sized `skbuffs` can affect Linux’s congestion control, unfortunately, which counts `skbuffs` sent instead of bytes. Section 8.1 discusses the effects of this Linux-specific issue, which a future version of *uTCP* will address.

The *uCOBS* Library: The *uCOBS* prototype is a user-space library in C, amounting to ~700 lines of code [15]. *uCOBS* presents simple `cobs_sendmsg()` and `cobs_recvmsg()` interfaces enabling applications to send and receive COBS-encoded datagrams, taking advantage of send-side prioritization and out-of-order reception depending on the presence of send- and receive-side OS support for *uTCP*, respectively.

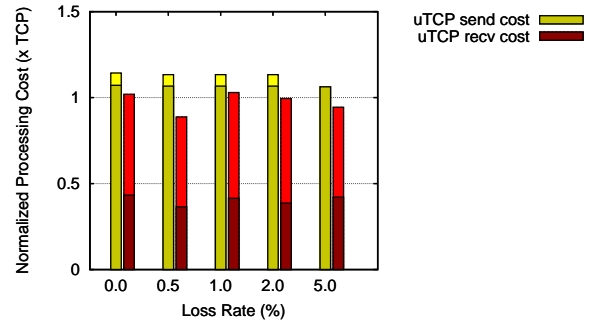


Figure 7: CPU costs of application with and without ofo-send modifications at different loss rates.

A *uTLS* Prototype Based on OpenSSL: The *uTLS* prototype builds on OpenSSL 1.0.0 [36], adding ~550 lines of code and modifying ~40 lines [15]. Applications use OpenSSL’s normal API to create a TLS connection atop a TCP socket, then set a new *uTLS*-specific socket option to enable out-of-order, record-oriented delivery on the socket. OpenSSL 1.0.0 unfortunately does not yet support TLS 1.1, the first TLS version that uses explicit Initialization Vectors (IVs), permitting out-of-order decryption. For experimentation, therefore, the *uTLS* prototype modifies OpenSSL’s TLS 1.0 ciphersuite to use explicit IVs as in TLS 1.1 would. Since this change breaks OpenSSL’s interoperability, our prototype is not suitable for deployment. We expect no major difficulties in porting *uTLS* to the next major OpenSSL release, which will support TLS 1.1.

8 Performance Evaluation

This section evaluates Minion via experiments designed to approximate realistic application scenarios. All experiments were run across three Intel PCs running Linux 2.6.34: between two machines representing end hosts, a third machine interposes on the path and uses `dumynet` [9] to emulate various network conditions. To minimize well-known TCP delays fairly for both TCP and *uTCP*, we enabled Linux’s “low latency” TCP code path via the `net.ipv4.tcp_low_latency` sysctl, and disabled the Nagle algorithm.

8.1 Bandwidth and CPU Costs

We first explore *uTCP*’s costs, with or without record encoding and extraction via *uCOBS* and *uTLS*, for a 30MB bulk transfer on a path with 60ms RTT.

Raw *uTCP*: *uTCP*’s CPU costs at both the sender and the receiver, without application-level processing, are almost identical to TCP’s CPU costs, across a range of loss rates from 0% to 5% (Figure 7).

Figure 8 shows bandwidth achieved by raw *uTCP* and TCP, for different application `write()` sizes. When message size is a multiple of TCP’s Maximum Segment

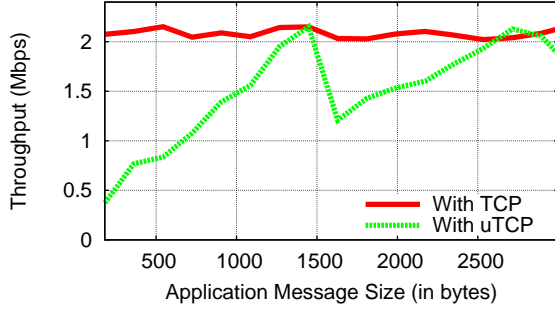


Figure 8: Throughput with different app message sizes.

Size (MSS)—at 1448 bytes (1 MSS) and 2896 bytes (2 x MSS)—*u*TCP’s throughput is the same as TCP’s.

The disparity elsewhere is due to Linux’s congestion control counting *skbuffs* instead of bytes, mentioned earlier in Section 7. Since *u*TCP does not pack all *skbuffs* to MSS size, for a given number of *skbuffs* *u*TCP transmits fewer bytes than TCP, reducing throughput.

While we believe that the correct way to fix this negative interference is to change the Linux congestion control mechanisms to be byte-based, as is done in BSD for instance, this is a particular implementation artifact that we do not address further in this paper. In our experiments, we therefore use MSS-sized (1448-byte) application messages where possible; where it is unavoidable, our results include this throughput penalty when the *u*TCP sender is used, and should improve when this interference is eliminated.

Costs with *u*COBS/*u*TLS: To measure these CPU costs, we run a 30MB bulk transfer over a path with a 60ms RTT, for several loss rates.

Figure 9(a) shows CPU costs including application-level encoding/decoding, atop standard TCP (“COBS”) and atop *u*TCP (“*u*COBS”), for several loss rates at both the sender and at the receiver. The lighter part of each bar representing user time and the darker part representing kernel time. These results are normalized to the performance of raw TCP, with no application-level encoding or decoding.

COBS encoding/decoding barely affects kernel CPU use but incurs some application-level CPU cost, which is a function of the encoding itself, and of the fact that the libraries can be significantly optimized further.

Figure 9(b) shows the CPU costs of *u*TLS relative to TLS. At the sender, the CPU costs are identical, since there is nothing that *u*TLS does differently than TLS, and since the CPU cost of using *u*TCP is practically the same as with TCP. The user-space cost for the *u*TLS receiver is generally higher than TLS, since the *u*TLS receiver does more work in processing out-of-order frames than

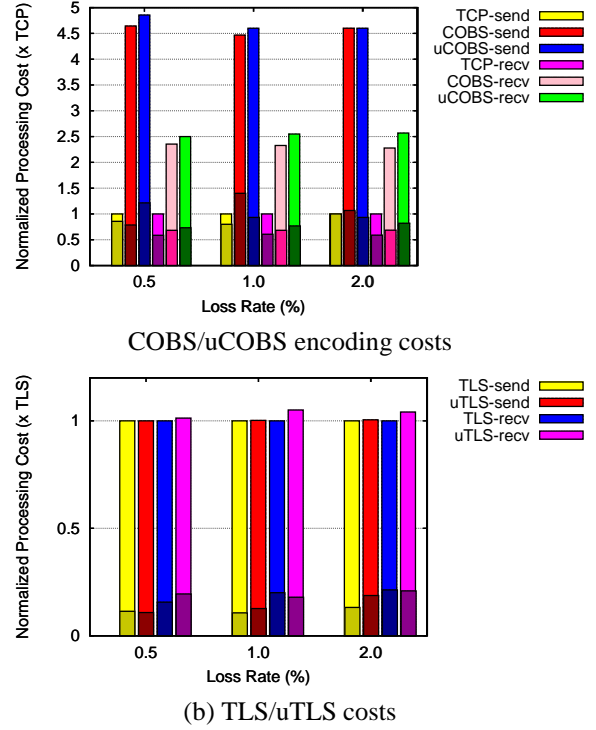


Figure 9: CPU costs of using an application with TCP, COBS, and *u*COBS at different loss rates.

the TLS receiver, but this cost remains within 7% of the TLS receiver’s cost.

The bandwidth penalty of *u*COBS encoding is barely perceptible, under 1%. TLS’s bandwidth overhead, up to 10%, is due to TLS headers, IVs, and MACs; *u*TLS adds no bandwidth overhead beyond standard TLS.

8.2 Conferencing Applications

We now examine a real-time Voice-over-IP (VoIP) scenario. A test application uses the SPEEX codec [53] to encode a WAV file using ultra-wideband mode (32kHz), for a 256kbps average bit-rate, and transmit voice frames at fixed 20ms intervals. Network bandwidth is 3Mbps and RTT is 60ms, realistic for a home broadband connec-

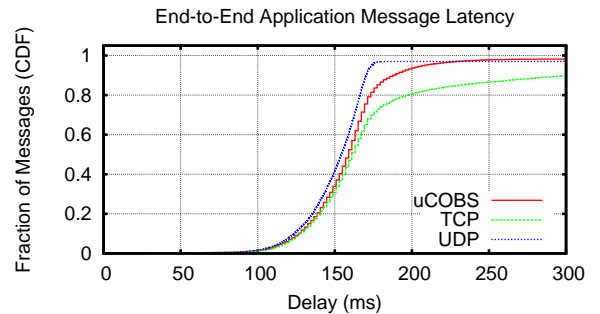


Figure 10: CDF of end-to-end latency in VoIP frames.

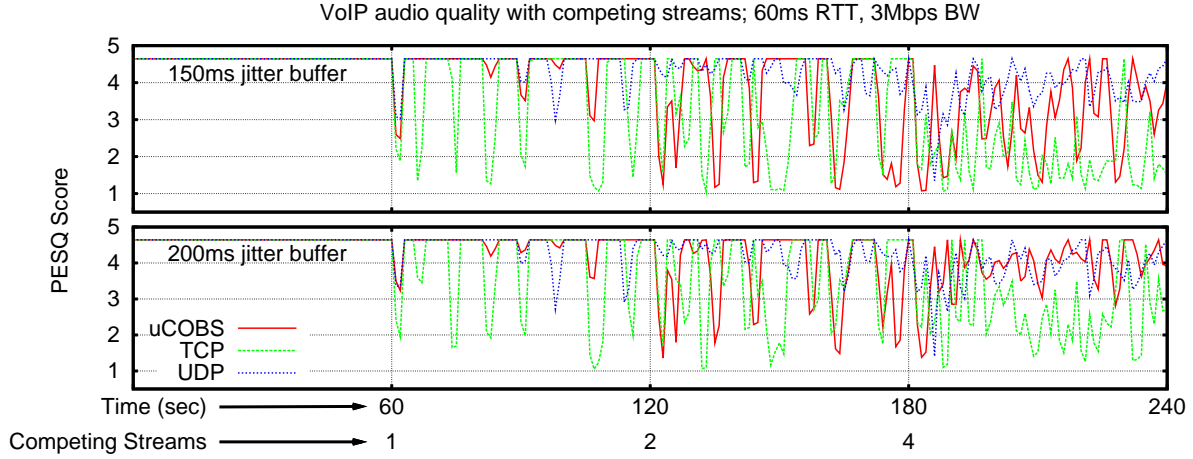


Figure 12: Moving PESQ score of VoIP call under increasing bandwidth competition.

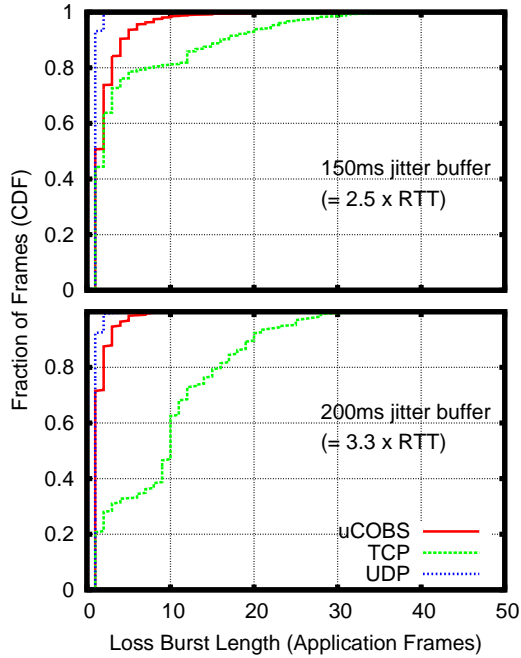


Figure 11: CDF of codec-perceived loss-burst size with TLV encoded frames over TCP, UDP, and uCOBS.

tion. To generate losses more realistically representing network contention, we run a varying number of competing TCP file transfers, emulating concurrent web browsing sessions or a BitTorrent download, for example.

Latency: Figure 10 shows a CDF of one-way per-frame latency perceived by the receiving application, under heavy contention from 4 competing TCP streams. All three transports suffer major delays. 4% of UDP frames do not arrive at all, since UDP does not retransmit. 95% of frames sent with *uCOBS* over *uTCP* arrive within 200ms, compared to 80% of TCP frames.

Burst Losses: VoIP codecs such as SPEEX can interpolate across one or two missing frames, but are sensitive to burst losses or delays, which yield user-perceptible blackouts. An application’s susceptibility to blackouts depends on its jitter buffer size: a larger buffer increases the receiver’s tolerance of burst losses or delays, but also increases effective round-trip delay, which can add user-perceptible “lag” to all interactions.

The CDF in Figure 11 shows the prevalence of different lengths of burst losses experienced by the receiver in a typical VoIP call. A burst loss is a series of consecutive voice frames that miss their designated playout time, due either to loss or delay. A 150ms jitter buffer and 4 competing streams atop TCP appears similar to *uCOBS* and UDP, but this is misleading. TCP has $3\times$ as many *total bursts* with a 150ms jitter buffer as with 200ms, with most of the extra bursts only a single packet. If these single dropped packets are kept by a larger jitter buffer, as in the bottom graph, TCP shows a much bleaker picture.

A 200ms jitter buffer of $3\times$ the path RTT might seem generous, but the ITU’s recommended maximum transmission time of 400ms [3] allows for a larger buffer with these network conditions. Now the differences between *uCOBS* and TCP are quite pronounced, with 80% of burst losses atop *uCOBS* being 3 or fewer packets, nearly

matching that of UDP. Meanwhile 40% of TCP's bursts are greater than 10 packets, producing highly-perceptible 1/5-second pauses.

Perceptual Audio Quality: The next experiment shows objectively that the characteristics demonstrated above for TCP, *u*TCP and UDP directly affect the perceived audio quality of a VoIP call. The Perceived Sound Quality (PESQ) [35] is an industry-recognized algorithm for objectively measuring the quality of a degraded sound file compared to a target sound file. In this work, we use the freely available International Telecommunication Union (ITU) reference implementation [34] for 862.2 PESQ score. PESQ scores range from roughly 1 (degraded audio is inaudible) to 4.644 (perfect match, no degradation).

We transmit a 4 minute VoIP call using the same 60ms RTT and 3Mbps link as before, with two jitter buffer sizes. Competing streams begin at 1-minute intervals, eventually causing significant degradation of audio quality with 4 competing streams.

We compare the received audio files to an "ideal" audio file with no dropped frames. Each data point represents the PESQ score for the next 2 seconds. We arbitrarily choose 2 seconds because we feel this is a reasonable amount of time a user considers when answering the question: "How is audio quality now?"

Our testing shows that audio quality degrades immediately once competing streams begin. Our goal is to show the user experience throughout the call as competing streams choke out the bandwidth. In such a scenario, we expect to see the audio quality rebound after the initial collision with competing TCP streams, which themselves back off due to their losses. Thus, Figure 12 simulates a more sustained user experience over the course of a 4 minute call, and represents a typical result for this experiment.

We see that even 1 competing stream causes significant drops for TCP, while *u*COBS and UDP exhibit only minor dips in quality. In the third minute of the call (2 competing streams), *u*COBS shows volatility compared to UDP. We attribute this to the congestion control of *u*COBS and TCP. The competing streams themselves use TCP's backoff mechanism, interleaving windows of high and low bandwidth for the VoIP call. *u*COBS still suffers from this less than TCP, however, due to its ability to deliver data out-of-order. Meanwhile, TCP's decreasing fidelity beyond 1 competing stream makes it potentially a non-option under any bandwidth competition.

Perhaps the most important feature shown by these experiments is the effect of the jitter size on *u*COBS, and the lack of effect for TCP. Increasing the jitter buffer from 150 to 200ms, particularly under 4 competing streams, improves *u*COBS significantly more than it

improves TCP. This speaks to the fundamental difference between the two highlighted in Figure 11: that TCP's bursty losses are ill-suited to real-time applications. Because a single lost packet delays subsequent packets, repeatedly dropped packets under heavy bandwidth contention are helpless with any reasonable jitter buffer size. Breaking the in-order requirement of delivery is essential to transmission quality in these situations, making *u*COBS more suitable for real-time applications. Indeed, increasing the jitter buffer size to 200ms brings *u*COBS in-line with UDP under 4 competing streams.

8.3 VPN Tunneling

Applications running atop TCP-based VPN tunnels often encounter *TCP-in-TCP* effects [51]. The applications' tunneled TCP flows assume they are running atop a best-effort, packet-switched network as usual, but are in fact running atop a reliable, in-order TCP-based tunnel. The TCP tunnel affects the tunneled flows' congestion control by increasing observed latency and RTT variance, and masks losses: tunneled flows never see "lost" or "re-ordered" TCP segments, only long-delayed ones. While *u*TCP does not change TCP's reliability or congestion control, it offers tunneled flows lower delay and jitter, and a more accurate view of packet losses.

To test Minion's impact on TCP-in-TCP effects, we made two changes to OpenVPN 2.1.4 [37]. First, we modified OpenVPN to use *u*COBS instead of TCP, enabling unordered delivery of tunneled IP packets. Second, to reduce delay variance of tunneled TCP flows further, the modified OpenVPN gives tunneled TCP ACKs a higher priority than other packets.

The experiment uses a link with 3Mbps download and 0.5Mbps upload bandwidth, consistent with the median speed of residential connections [57]. We measure the performance of a TCP connection representing a download through a VPN tunnel that also carries competing TCP connections in the upload direction. ACKs for the measured TCP flow thus compete for bandwidth with data from the competing flows.

Figure 13 shows measured throughput of the download, with TCP and *u*TCP, for a varying number of competing uploads. While *u*TCP does not eliminate all TCP-in-TCP effects, the reduced RTT and RTT-variance often noticeably improved performance.

8.4 Multistreaming Web Transfers

We now explore building concurrency atop *u*TCP's unordered message service; we build a *multistreaming* abstraction that provides multiple independent and ordered message streams *within* a single *u*TCP connection. While both the need for and the benefits of concurrency at the transport layer have been well known [19, 22, 33,

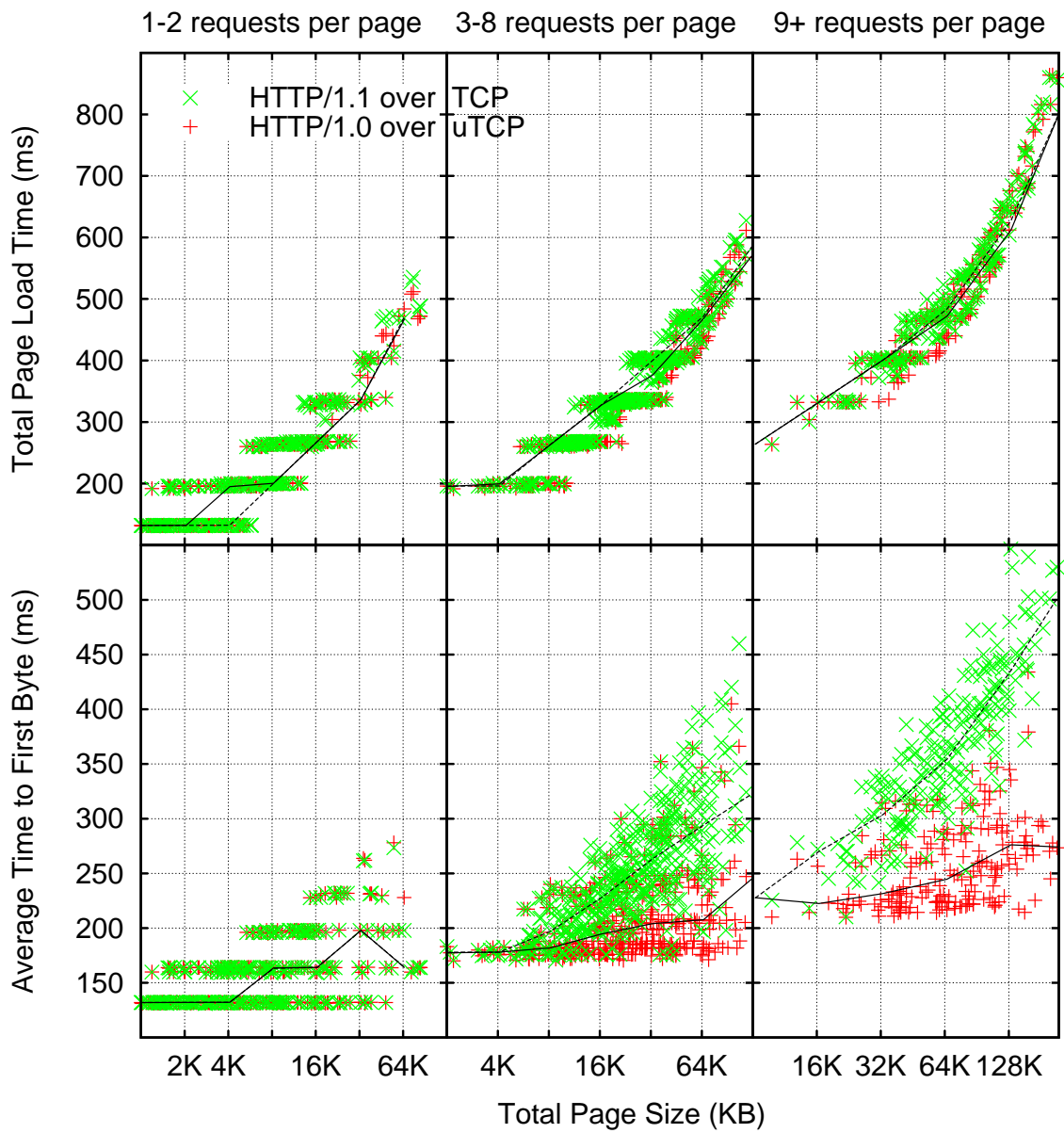


Figure 14: Pipelined HTTP/1.1 over a persistent TCP connection, vs. Parallel HTTP/1.0 over *ms*TCP.

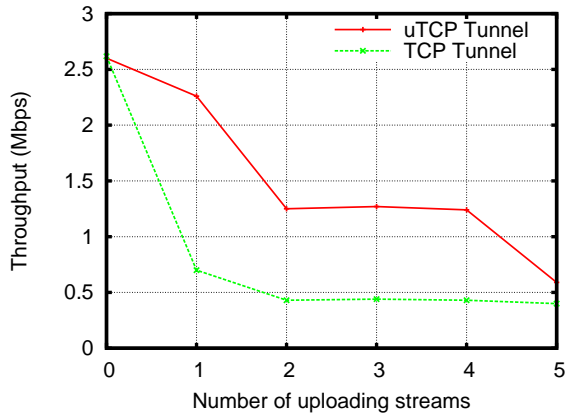


Figure 13: Throughput obtained by a TCP flow through a TCP/uTCP VPN.

48], to our knowledge, this is the first time that true multistreaming has been built using TCP.

Our prototype implementation of a multistreaming library in user-space atop *uTCP*, which we call *Multistreamed TCP* or *msTCP*, uses a 16-bit stream identifier, and a 16-bit stream sequence number to order chunks within a stream. An application specifies a stream identifier when using *msTCP*’s `ms_sendmsg()` API; *msTCP* then breaks every application message down into a series of *chunks*—the *msTCP*-defined smallest multiplexable unit of data within a connection. Chunks are prepended with a 16-byte chunk header, which includes the stream identifier for the message and the stream sequence number, and then finally transmitted using *uCOBS*. At the receiver, *msTCP*’s `ms_recvmsg()` receives unordered chunks from the underlying *uCOBS*, parses the chunk header, and adds each chunk to its stream’s *stream-queue*—`ms_recvmsg()` maintains one queue for each stream identifier that it encounters in the received chunks—the chunks are then delivered to the application in-order within their respective streams. Note that *msTCP*’s API can be trivially extended to using *uTLS* as well.

We use the Web as an example application to illustrate the behavior of *msTCP*. HTTP/1.1 addressed the inefficiency of short-lived TCP streams through pipelining over persistent connections. Since *msTCP* attempts to offer the benefits of persistent streams with the simplicity of the one-transaction-per-stream model, We now compare the performance of HTTP/1.0 with parallel object requests over *msTCP* against the behavior of pipelined HTTP/1.1 over a persistent TCP connection, under a simulated web workload.

For this test we simulate a series of web page loads, each page consisting of a “primary” HTTP request for the HTML, followed by a batch of “secondary” requests for embedded objects such as images. As the simula-

	TCP	<i>uTCP</i>	DCCP	SCTP
Kernel Code	12,982	565 (4.6%)	6,338	19,312
	<i>uCOBS</i>	SSL/TLS	<i>uTLS</i>	DTLS
User Code	732	31,359	586 (1.9%)	4,734

Table 1: Code size of *uTCP* prototype as a delta to Linux’s TCP stack, the *uCOBS* library, and *uTLS* as a delta to `libssl` from OpenSSL. Code sizes of “native” out-of-order transports are included for comparison.

tion’s workload we use a fragment of the UC Berkeley Home IP web client traces available from the Internet Traffic Archive [27]. We sort the trace by client IP address so that each user’s activities are contiguous, then we use only the order and sizes of requests to drive the simulation, ignoring time stamps. Since the traces do not indicate which requests belong to one web page, the simulation approximates this information by classifying requests by extension into “primary” (e.g., `.html` or no extension) and “secondary” (e.g., `.gif`, `.jpg`, `.class`), and then associating each contiguous run of secondary requests with the immediately preceding primary request. The simulation pessimistically assumes that the browser cannot begin requesting secondary objects until it has downloaded the primary object completely, but at this point it can in theory request all of the secondary objects in parallel. The experimental setup uses a link with 1.5Mbps bandwidth in each direction and with a 60ms RTT, typical of browsing over a residential connection.

Figure 14 shows a scatter-plot of total time to load the entire page in the top three graphs, and the bottom three graphs show the average time to load the first byte of an object within each page—the expected time for an object to start being rendered at the browser. The X-axis represents total webpage size. The dark curves show median times, computed across webpages in log-sized buckets of webpage size—the solid curve shows the median for HTTP/1.1 over TCP and the dashed curve for HTTP/1.0 over multistreaming.

The total time to transfer the pages remains the same; *msTCP*’s bit-overheads do not affect application-observed throughput noticeably. *msTCP* however, shows much lower latency in loading the first byte of objects, since the objects’ chunks are interleaved within the persistent connection by *msTCP*.

These latency savings in *msTCP* should benefit web frameworks like SPDY [1] and more efficiently enable the use of HTTP/HTTPS as a substrate for deploying latency-sensitive applications [39].

8.5 Implementation Complexity

To evaluate the implementation complexity of *u*TCP and the related application-level code, Table 1 summarizes the source code changes *u*TCP makes to Linux's TCP stack in lines of code [15], the size of the standalone *u*COBS library, and the changes *u*TLS makes to OpenSSL's `libssl` library. The SSL/TLS total does not include OpenSSL's `libcrypto` library, which `libssl` requires but *u*TLS does not modify.

With only a 600-line change to the Linux kernel and less than 1400 lines of user-space support code, *u*TCP provides a delivery service comparable to Linux's 6,300-line native DCCP stack, while providing greater network compatibility. In user space, *u*TLS represents less than a 600-line change to the stream-oriented SSL/TLS protocol, contrasting with OpenSSL's 4,700-line implementation of DTLS, which runs only atop out-of-order transports such as UDP or DCCP.

9 Related Work

New transports for latency-sensitive apps: Brosh et al. [8] model TCP latency, and identify the regions of operation for latency-sensitive apps with TCP. While some of the considerations apply, such as latency induced by TCP congestion control, *u*TCP extends the working region for such apps by eliminating delays at the receiver.

DCCP [30, 31] provides an unreliable, unordered datagram service with negotiable congestion control. SCTP [48] provides unordered and partially-ordered delivery services to the application. Both DCCP and SCTP face large deployment barriers on today's Internet, however, and are thus not widely used.

New transports such as SST [19] and CUSP [50] run atop UDP to increase deployability, and UDP tunneling schemes have been proposed for standardized Internet transports as well [38, 52]. Many Internet paths block UDP traffic as well, however, as evidenced by the shift of popular VoIP applications such as Skype [6] and VPNs such as DirectAccess [16] toward tunneling atop TCP instead of UDP, despite the performance disadvantages.

Message Framing over TCP: Protocols such as HTTP [18], SIP [45], and iSCSI [46], can all benefit from out-of-order delivery, but use TCP for legacy and network compatibility reasons. All use simple type-length-value (TLV) encodings, which do not directly support out-of-order delivery even with *u*TCP, because they offer no reliable way to distinguish a record header from data in a TCP stream fragment. While COBS [12] represents an attractive set of characteristics for framing records to enable out-of-order delivery, other encodings such as BABS [10] also represent viable alternatives.

10 Conclusion

For better or worse, TCP remains the most common substrate for application-level protocols and frameworks, many of which can benefit from unordered delivery. Minion demonstrates that it is possible to obtain unordered delivery from wire-compatible TCP and TLS streams with surprisingly small changes to TCP stacks and application-level code. Without discounting the value of UDP and newer OS-level transports, Minion offers a more conservative path toward the performance benefits of unordered delivery, which we expect to be useful to applications that use TCP for a number of pragmatic reasons.

Acknowledgments

This research is sponsored by the NSF under grants CNS-0916413 and CNS-0916678.

References

- [1] SPDY: An Experimental Protocol For a Faster Web. <http://www.chromium.org/spdy/spdy-whitepaper>.
- [2] "zeromq: The intelligent transport layer". <http://www.zeromq.org>.
- [3] International Telecommunication Union. Series g. 114 one-way transmission time, May 2003.
- [4] F. Audet, ed. and C. Jennings. Network address translation (NAT) behavioral requirements for unicast UDP, Jan. 2007. RFC 4787.
- [5] H. Balakrishnan, H. S. Rahul, and S. Seshan. An integrated congestion management architecture for Internet hosts. In *ACM SIGCOMM*, Sept. 1999.
- [6] S. A. Baset and H. Schulzrinne. An analysis of the Skype peer-to-peer Internet telephony protocol. In *IEEE INFOCOM*, Apr. 2006.
- [7] L. Brakmo and L. Peterson. TCP Vegas: End to end congestion avoidance on a global Internet. *IEEE Journal on Selected Areas in Communications*, 13(8):1465–1480, Oct. 1995.
- [8] E. Brosh, S. A. Baset, V. Misra, D. Rubenstein, and H. Schulzrinne. The delay-friendliness of tcp for real-time traffic. *IEEE/ACM Trans. Netw.*, 18(5):1478–1491, 2010.
- [9] M. Carbone and L. Rizzo. Dummynet Revisited. *ACM CCR*, 40(2), Apr. 2010.
- [10] J. S. Cardoso. Bandwidth-efficient byte stuffing. In *IEEE ICC 2007*, 2007.
- [11] B. Carpenter and S. Brim. Middleboxes: Taxonomy and Issues, Feb. 2002. RFC 3234.
- [12] S. Cheshire and M. Baker. Consistent Overhead Byte Stuffing. In *ACM SIGCOMM*, Sept. 1997.
- [13] Cisco. Rate-Based Satellite Control Protocol, 2006.
- [14] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. In *ACM SIGCOMM*, pages 200–208, 1990.

- [15] A. Danial. Counting Lines of Code, ver. 1.53. <http://cloc.sourceforge.net/>.
- [16] J. Davies. DirectAccess and the thin edge network. *Microsoft TechNet Magazine*, May 2009.
- [17] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol version 1.2, Aug. 2008. RFC 5246.
- [18] R. Fielding et al. Hypertext transfer protocol — HTTP/1.1, June 1999. RFC 2616.
- [19] B. Ford. Structured streams: a new transport abstraction. In *ACM SIGCOMM*, Aug. 2007.
- [20] B. Ford and J. Iyengar. Breaking up the transport logjam. In *7th Workshop on Hot Topics in Networks (HotNets-VII)*, Oct. 2008.
- [21] B. Ford and J. Iyengar. Efficient cross-layer negotiation. In *8th Workshop on Hot Topics in Networks (HotNets-VIII)*, Oct. 2009.
- [22] J. Gettys and H. F. Nielsen. SMUX Protocol Specification, July 1998. <http://www.w3.org/TR/WD-mux>.
- [23] S. Guha, Ed., K. Biswas, B. Ford, S. Sivakumar, and P. Srisuresh. NAT behavioral requirements for TCP, Oct. 2008. RFC 5382.
- [24] L. Guo, E. Tan, S. Chen, Z. Xiao, O. Spatscheck, and X. Zhang. Delving into Internet Streaming Media Delivery: a Quality and Resource Utilization Perspective. In *IMC*, Oct. 2006.
- [25] M. Herbert. Noq: back-pressured/blocking ip networks, Jan. 2005. <http://marc.herbert.free.fr/noq/>.
- [26] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it still possible to extend TCP? In *Internet Measurement Conference*, Nov. 2011.
- [27] The Internet traffic archive. <http://ita.ee.lbl.gov/>.
- [28] D. Katabi, M. Handley, and C. Rohrs. Internet congestion control for high bandwidth-delay product networks. In *ACM SIGCOMM*, Aug. 2002.
- [29] S. Kent and K. Seo. Security architecture for the Internet protocol, Dec. 2005. RFC 4301.
- [30] E. Kohler, M. Handley, and S. Floyd. Datagram congestion control protocol (DCCP), Mar. 2006. RFC 4340.
- [31] E. Kohler, M. Handley, and S. Floyd. Designing DCCP: Congestion control without reliability. In *ACM SIGCOMM*, 2006.
- [32] J. Mogul. TCP Offload is a Dumb Idea Whose Time Has Come. In *HotOS IX*, May 2003.
- [33] P. Natarajan et al. SCTP: An innovative transport layer protocol for the web. In *15th World Wide Web Conference (WWW)*, May 2006.
- [34] I.-T. T. S. S. of ITU. Recommendation p.862 (2001) amendment 2 (11/05) reference implementation and conformance testing, Nov. 2005. <http://www.itu.int/rec/T-REC-P.862-20051> protocol, July 1984. RFC 908.
- [35] I.-T. T. S. S. of ITU. Wideband extension to recommendation p.862 for the assessment of wideband telephone networks and speech codecs, Nov. 2007.
- [36] The OpenSSL project. <http://www.openssl.org/>.
- [37] The OpenVPN project. <http://openvpn.net/>.
- [38] T. Phelan. DCCP Encapsulation in UDP for NAT Traversal (DCCP-UDP), Aug. 2010. Internet-Draft draft-ietf-dccp-udpencap-02 (Work in Progress).
- [39] L. Popa, A. Ghodsi, and I. Stoica. HTTP as the narrow waist of the future Internet. In *9th ACM Workshop on Hot Topics in Networks (HotNets-IX)*, Oct. 2010.
- [40] J. Postel. User datagram protocol, Aug. 1980. RFC 768.
- [41] C. Reis et al. Detecting in-flight page changes with web tripwires. In *5th Symposium on Networked System Design and Implementation (NSDI)*, Apr. 2008.
- [42] E. Rescorla. HTTP over TLS, May 2000. RFC 2818.
- [43] E. Rescorla and N. Modadugu. Datagram transport layer security, Apr. 2006. RFC 4347.
- [44] J. Rosenberg. UDP and TCP as the new waist of the Internet hourglass, Feb. 2008. Internet-Draft (Work in Progress).
- [45] J. Rosenberg et al. SIP: session initiation protocol, June 2002. RFC 3261.
- [46] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. Internet small computer systems interface (iSCSI), Apr. 2004. RFC 3720.
- [47] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A transport protocol for real-time applications, July 2003. RFC 3550.
- [48] R. Stewart, ed. Stream control transmission protocol, Sept. 2007. RFC 4960.
- [49] Transmission control protocol, Sept. 1981. RFC 793.
- [50] W. W. Terpstra, C. Leng, M. Lehn, and A. P. Buchmann. Channel-based unidirectional stream protocol (CUSP). In *IEEE INFOCOM Mini Conference*, Mar. 2010.
- [51] O. Titz. Why TCP over TCP is a bad idea, Apr. 2001. <http://sites.inka.de/bigred/devel/tcp-tcp.html>.
- [52] M. Tuexen and R. Stewart. UDP Encapsulation of SCTP Packets, Jan. 2010. Internet-Draft draft-tuexen-sctp-udp-encaps-05 (Work in Progress).
- [53] J.-M. Valin. The speex codec manual version 1.2 beta 3, Dec. 2007. <http://www.speex.org/>.
- [54] D. Velten, R. Hinden, and J. Sax. Reliable data

- [55] N. Vratonjic, J. Freudiger, and J.-P. Hubaux. Integrity of the web content: The case of online advertising. In *Workshop on Collaborative Methods for Security and Privacy*, Aug. 2010.
- [56] W3C. The websocket api (draft), 2011. <http://dev.w3.org/html5/websockets/>.
- [57] www.speedmatters.org. 2010 report on internet speeds in all 50 states, Nov. 2010. <http://www.speedmatters.org/content/internet-speed-report>.
- [58] M. Zec, M. Mikuc, and M. Zagar. Estimating the Impact of Interrupt Coalescing Delays on Steady State TCP Throughput. In *SoftCOM*, 2002.